Feature Interactions:
A Mixed Semantic Model Approach

Paul Gibson
CRIN URA 262 du CNRS

Bruno Mermet
CRIN URA 262 du CNRS

Dominique Méry*
Université Henri Poincaré-Nancy 1 & Institut Universitaire de France

email: (gibson/mery/mermet)@loria.fr

May 2, 1997

Abstract

The feature interaction problem is prominent in telephone service development. Through a number of case studies, we have discovered that no one semantic framework is suitable for the synthesis and analysis of formal feature requirements models. We illustrate our mixed-model approach, where we use OO LOTOS, B and TLA+ in a complementary fashion. A simple combination of *call forwarding* and *call screening* features emphasises the need for different semantics during the analysis, synthesis, validation, refinement and verification stages of formal development.

1 Introduction

The feature interaction problem is stated simply, and informally, as follows: A *feature interaction* is a situation in which system behaviour (specified as some set of features[1]) does not as a whole satisfy each of its component features individually. We concentrate on the domain of telephone features [8, 5]. Figure 1 illustrates this definition within the formal framework which we adopt throughout this paper.

Three types of formal models are used. Firstly, we require an executable model (written in LOTOS [19] using an object-based style [16]) which is useful for constructing an executable model for validation. Secondly, we have a logical model (based on the B[1] method) which is

*This work is supported by the Institut Universitaire de France, by the HCM Scientific Network MEDICIS (CHRX-CT92-0054) and by the contract n°96 1B CNET-FRANCE-TELECOM & CRIN-CNRS-URA262

[1] In this paper we make no distinction between a feature and a service.

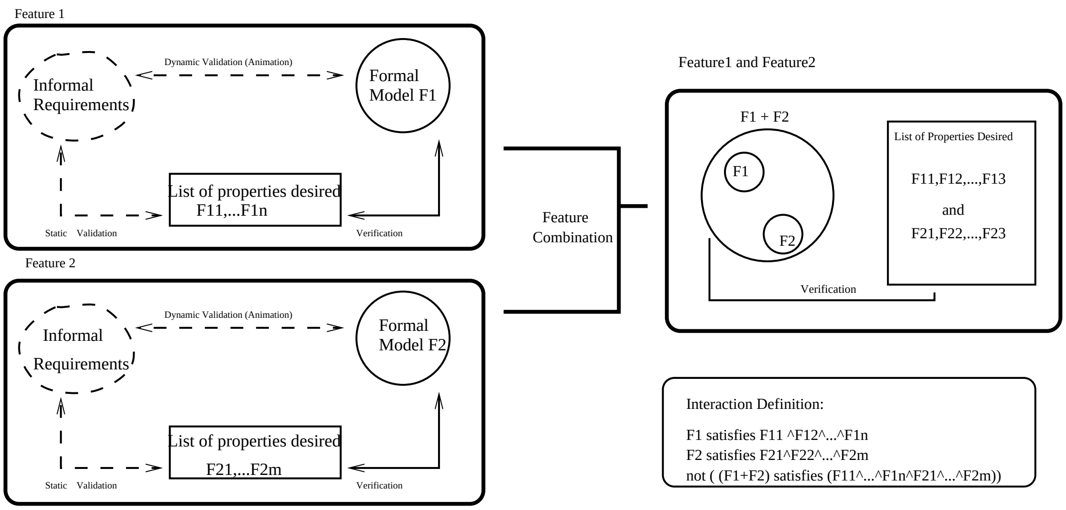

Figure 1: Feature Interaction: A formalisation

used to verify the state invariant properties of our system (statically). Finally, we use TLA [23] to provide semantics for a static analysis of liveness and fairness properties. No one model can treat each of these aspects, yet each of these aspects of the conceptualisation are necessary in the formal development of features.

2 Feature Interaction: What's so difficult?

Features are observable behaviour and are therefore a requirements specification problem [31]. Most feature interaction problems can be (and should be) resolved at the requirements capture stage of development. If there are no problems in the requirements specification then problems during the design and implementation will arise only through errors in the refinement process. Certainly the feature interaction problem is more prone to the introduction of such errors because of the highly concurrent and distributed nature of the underlying implementation domain, but this is for consideration *after* each individual feature's requirements have been modelled and validated; otherwise it will not be easy to identify the source of the interaction. Features are requirements modules *and* the units of incrementation as systems evolve. A telecom system is a set of features. A feature interaction occurs in a system whose complete behaviour does not satisfy the separate specifications of all its features. Having features as the incremental units of development is the source of our complexity:

Complexity explosion: Potential feature interactions increase exponentially with the number of features in the system.

Chaotic Information Structure In Sequential Development Strategies: The arbitrary

sequential ordering of feature development is what drives the internal structure of the resulting system. As each new feature is added, the feature must include details of how it is to interact with all the features already in the system. Consequently, to understand the behaviour of one feature, it is necessary to examine the specification of all the features in the system. All conceptual integrity is lost since the distribution of knowledge is chaotic.

Assumption Problem: Already developed features often rely on assumptions which are no longer true when later features are conceived. Consequently, features may rely on contradictory assumptions.

Independent Development: Traditional approaches require a new feature developer to consider how the feature operates with all others already on the system. Consequently, we cannot concurrently develop new features: since how the new features work together will not be considered by either of the two independent feature developers. We want to have an incremental approach in which the developers do not need to know anything about the other features in the system. In our approach, it is the system designers who must resolve the integration problems: integration methods arise from an analysis of the features to be integrated. Formal requirements of individual features are required for the integration process to be verified.

Interface Problem: User controls on traditional phones are very limited and hence the input and output signals become polymorphic. This is a major problem in requirements specifications as it can lead to the introduction of ambiguities in systems of features. Formal requirements models make explicit the mapping between abstract and concrete actions and our systems can be automatically verified to ensure an absence of ambiguity that could lead to interactions.

Invalid Plain Old Telephone Service (POTS) Assumptions: Phone systems have changed dramatically over the past ten years. Many people (including feature developers) are not aware of the underlying complexity in the concrete system and, as a way of simplifying the problem, often make incorrect assumptions based on their knowledge of the plain old telephone service.

The feature interaction problem is difficult: having formal requirements models makes it manageable. A formal model of requirements is unambiguous — there is only one correct way to interpret the behaviour being defined. Although the model must still be mapped onto the real world (i.e. validated by the customer), this mapping is in essence more rigorous than in informal approaches. Building a formal model requires a better understanding of the problem domain and a better understanding of how the problem domain is viewed by the customer. A formal model can explicitly model nondeterminism, when choice of behaviour is specified. Another advantage of a mathematical approach is that high levels of expressibility allow the definition of *what* rather than *how*. In our formal approach, interactions occur only when requirements of multiple features are *contradictory*. The complexity of understanding the problem is thus contained within a definition of *contradiction* in our semantic framework.

3 OO LOTOS

In the thesis of Gibson [17], which forms the basis of our development method, there is an object oriented semantics based upon a constructive, easy to understand, state transition system model. The abstract data type (ADT) part of LOTOS, see [6, 21, 29], is used to implement the requirements models which are defined using this semantics. Then, *full LOTOS* (LOTOS with the ADT and process algebra parts) is used to model the requirements in a more concrete framework, during design. LOTOS is advantageous during feature development because:

- Its natural division into ADT part (based on ACT ONE [13]) and process algebra part (similar to CSP [20] and CCS [26]) suits our need for semantic continuity from analysis to design.

- LOTOS is a wide spectrum language, which is suitable for specifying systems at various levels of abstraction. Consequently, it can be used at both ends of the design process.

- There is wide support, often in the form of tools, for the static analysis and dynamic execution of LOTOS specifications: for example, see [29, 19, 3, 27].

- The object oriented methodology, with all its associated advantages, is supported by the OO LOTOS work. This makes building and validating models much more compositional.

OO LOTOS does not offer us an easy means of verifying design steps as model refinements. Furthermore, the LOTOS semantics do not include the notion of liveness and fairness.

4 TLA

TLA+ [25] is a specification language based on set theory and the temporal logic of actions, TLA [24]. It is a state-based methodology such as VDM [22], Z [28] or B [1]. However, unlike these other methods, fairness or eventuality properties on traces can be expressed and verified. A system is modelled as a set of behaviours and a behaviour is built from the set of states according to the actions that lead from one state to another state. The TLA logic has been integrated with a specification language [25] based on set theory and is structured by modules. Actions $a_0, a_1 \ldots a_i, \ldots$ are perfomed by the system and an idle action corresponds to stuttering. A behaviour is a mathematical object that characterizes the real system and it is a way to interpret TLA formulae. The set of behaviours for S is denoted $Beh(S)$. A system S satisfies a formula Π, if any behaviour of S satisfies Π: $S \models \Pi$, if $\forall \sigma \in Beh(S) : \sigma \models \Pi$, where Π is a TLA formula.

TLA formula are based on the use of primed and unprimed variables: a primed variable designates the state of the unprimed variable in the next state. For instance, $x' = x + 1$

will mean that x is incremented by one. Moreover, Lamport combines this new kind of basic assertion with temporal operators and the stuttering principle. A TLA formula has one of the following forms (where: P is a predicate[2], f is a state function, $\mathcal{A}$ is an action, x is a variable, and F and G are TLA formulae):

P Satisfied by a behaviour iff P is true for the initial state.

$\Box[\mathcal{A}]_f$ Satisfied by a behaviour iff every step satisfies $\mathcal{A}$ or leaves f unchanged.

$\Box F$ (F is always true.) Satisfied by a behaviour iff F is true for all suffixes of the behaviour.

$\exists\, x : F$ Satisfied by a behaviour iff there are some values that can be assigned to x to produce a behaviour satisfying F.

$\mathrm{WF}_f(\mathcal{A})$ (Weak fairness of $\mathcal{A}$) Satisfied by a behaviour iff $\mathcal{A} \wedge (f' \neq f)$ is infinitely often not enabled, or infinitely many $\mathcal{A} \wedge (f' \neq f)$ steps occur.

$\mathrm{SF}_f(\mathcal{A})$ (Strong fairness of $\mathcal{A}$) Satisfied by a behaviour iff $\mathcal{A} \wedge (f' \neq f)$ is only enabled finitely often, or infinitely many $\mathcal{A} \wedge (f' \neq f)$ steps occur.

$\Diamond F$ (F is eventually true) Defined to be $\neg\Box\neg F$.

$F \leadsto G$ (Whenever F is true, G will eventually become true) Defined to be $\Box(F \Rightarrow \Diamond G)$.

Using TLA has two advantages. Firstly, the semantics of eventuality and fairness are easy to exploit. Secondly, refinement is proved by logical implication. Unfortunately, TLA does not provide a modular approach to model construction. Furthermore, TLA is not easily validated by the customer.

5 The B Method

The B method is a formal specification method developed by Abrial [1]. It is based on set theory and first order predicate logic. B specifications are structured as *machines*. A machine encapsulates variables and operations on these variables. Variables can have any type of the Zermelo Set Theory. Each machine has an invariant, that describes a set of states in which its variables must remain. The method allows an automatic generation of *proof obligations*: theorems that must be proved to guarantee that the machine has the following properties :

- its initialisation establishes the invariant ;
- each operation preserves the invariant.

[2]Having Boolean predicates means that we include Boolean operators with their usual meanings.

Moreover, the B Method allow a *data refinement* process : an abtract machine can be refined by another. The method gives rules to prove the correctness of the refinement. Once again, the proof obligations can be generated automatically. The refinement process allows a progressive transition from an abstract high-level specification to a concrete low-level specification. If the last refinement is completly deterministic, then an automatic translation into a programming language can be done. The fact that many steps of a formal development in B can be done automatically makes this method well suited to an industrial usage. Two tools are currently available:

- the Atelier B, developed by Steria[12] ;
- the BToolkit, developed by B-Core[2]

Both tools are based on a theorem prover that solves between 60% and 80% of the proof obligations generated, the rest have to be analysed interactively.(These percentages are taken from a small subset of case studies. Currently, we are extending the set of data and hope to publish some more *meaningful* results. The complete set of data is available on request.)

The strength of the B method is in the specification and verification of invariant properties, and the preservation of these properties during refinment. Although there are some compositional operators in B, these are not as powerful as those associated with our OO conceptualisation as they are primarily syntactic (like macro expansions). There is an animator built into the B tools but, again, the validation process would benefit from more powerful composition operators. Finally, unlike TLA, B does not provide semantics for fairness and liveness.

6 The Plain Old Telephone Service (POTS)

As an aid to comprehension we limit our domain of study to interactions which deal only with the telephone users' points of view. We also restrict our level of abstraction: problems due to sharing of resources, information hiding, interface realisation, etc . . . are design issues and are not examined here. The key to understanding features is the language(s) we employ for communication, verification and validation. Three different, though complementary, views of features play a role in the process of requirements capture. Before we examine the specific interaction of the *call screening* and *call forwarding* features, we summarise the different semantic views and use them to model the POTS requirements of a single phone[3].

[3] This specification is much simpler than the real case, but we abstract away from a number of details in order to simplify our reasoning.

States and Actions: A dynamic (object LOTOS) view

Labelled state transition systems are often used to provide executable models during the analysis and requirements stages of software development [10, 11]. In particular, such models play a role in many of the object oriented analysis and design methods [4, 9]. However, a major problem with state models is that it can be difficult to provide a good decomposition of large, complex systems when the underlying state and state transitions are not fully understood. The object oriented[4] paradigm provides a natural solution to this problem. By equating the notion of class with the state transition system model, and allowing the state of one class to be defined as a composition of states of other classes, we provide a means of specifying state transition models in a constructive fashion. Further, such an approach provides a more constructive means of testing actual behaviour against requirements. This is explained in more detail in [17] where OO LOTOS is used to provide a state based view of objects as processes.

The equations of a class specify how all its member objects respond to all valid service requests. In fact, they correspond to labelled state transitions. The equations map a state before the transition to a state after transition (and may be labelled by a value to be returned to the service requester). An object's state can be structured or unstructured. An unstructured state is represented by a literal value. A structured state is represented by a set of component objects. Composition is primarily a relationship between objects. It can, however, be extended to classes: when all class *members* are represented by the same *structure* then the class can, without ambiguity, be said to be *composed* from the classes which parameterise the *structure*.

The dynamic behaviour is modelled by the state transition system, corresponding to our OO LOTOS specification, in figure 2.

This state based object view forms the basis on which we build our feature animations and permit behaviour validation in a compositional manner. However, they are not *good* for formal reasoning about feature requirements. For this we need to consider specification of state invariants and fairness properties.

Defining Invariants: A static (B) view

Invariants are needed to specify abstract properties of a system which must always be true. The object oriented approach permits the definition of three sorts of invariant:

- **Typing:** By stating that all objects are defined to be members of some class we are in fact specifying an invariant. This invariant is verified automatically by the O-LSTS tools.
- **Service requests:** Typing also permits us to state that objects in our system will only ever be asked to perform services that are part of their interfaces. This invariant is also

[4] In fact, in this paper we do not consider such issues such as subclassing, inheritance and polymorphism; thus the view is really object based.

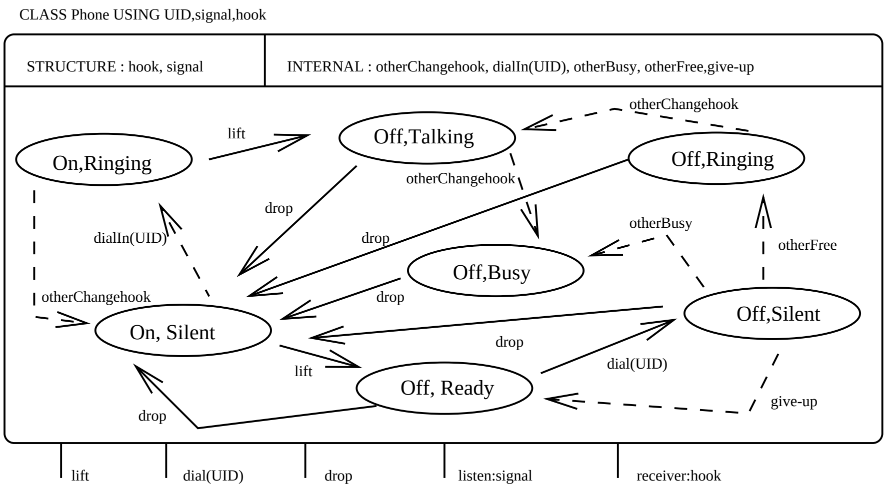

Figure 2: A single POTS phone user model

verified automatically by the O-LSTS tools.

- **State Component Dependencies:** In a structured class we may wish to specify some property that depends on the state of two or more of the components, and which is always true. This cannot be statically verified using the O-LSTS semantics (of OO LOTOS) but can be treated through a dynamic analysis (model check). Unfortunately, such a model check cannot be guaranteed when we have large (possibly infinite) numbers of states in our systems. For this reason we need to adopt a more proof theoretic framework (such as the B tool).

By translating our state invariant requirements into another proof theoretic framework[5] we have been able to statically verify our state component invariants. These invariants are often the key to feature interactions: when features are components and their invariants are not guaranteed by the containing system then we have an interaction.

A state invariant which defines a relation between the hook component and the signal component will include the following requirement: `(signal = talking) => (hook = off)`. In the simple finite telephone system, this is directly verifiable by checking that it is true in all the states. In the network of many different telephones we use the invariant to specify relations between pairs of phones. For example, a simple POTS state invariant requirement is that `only an even number of phones can be talking at the same time`. This is proved by showing that it is true in the initial state (where all phones are `off` and `ready`). Then we show that all possible state transitions maintain the required

[5]In fact, we have successfully performed the translation to B and PVS.

property (if it is true before the action occurs then it is true after the action occurs). This property cannot be checked through an exhaustive search of a system of an unbounded number of phones. It can be checked by proving that all transitions are closed with respect to the invariant.

Temporal Logics: A fairness (TLA) view

In TLA, a system is modeled as a set of traces over a set of states. The specifier may decide to ignore traces that do not satisfy a scheduling policy, such as strong or weak fairness, and temporal operators, such as $\Box$ (Always) or $\Diamond$ (Eventually), are combined to express these assumptions over the set of traces. Such fairness is important in feature specification and cannot be easily expressed using OO LOTOS or B. The key is the need for different abstractions for nondeterminism in our requirements models.

Without a temporal logic, nondeterminism in the features can be specified only at one level of abstraction: namely that of an internal choice of events. This can lead to many problems in development. For example, consider the specification of a shared database. This database must handle multiple, parallel requests from clients. The order in which these requests are processed is required to be nondeterministic. This is easily specified in our OO semantic framework. However, the requirements are now refined to state that every request must be eventually served (this is a fairness requirement which we cannot express in our semantic framework). The only way this can be done is to *over-specify* the requirement by defining how this fairness is to be achieved (for example, by explicitly queueing the requests). This is bad because we are enforcing implementation decisions at the requirements level. With TLA we can express fairness requirements without having to say how these requirements are to be met.

Nondeterminism in the telephone is specified through internal events. These correspond to actions which are not available to the phone user but which perform state transitions. The simple phone example illustrates the need for fairness (where we specify that something good eventually happens rather than that something bad never happens). Consider a telephone which has just dialled a number and is off hook and silent. The user does not wish to remain in this state indefinitely but can only drop the phone if they wish to instigate a state change. By specifying fairness on the `give-up` action we guarantee that this state transition must eventually happen if the phone stays in the off-silent state (for whatever reason).

A TLA specification of a more complicated version of POTS is given below. It illustrates the use of fairness to guarantee the eventuality of internal events.

module *POTS-SPECIFICATION*

This module defines the specification of the Basic Phone System POTS and it uses modules CALL-DEFINITIONS, POTS-ACTIONS-OFF-HOOK, POTS-ACTIONS-ON-HOOK, POTS-ACTIONS-COMMUNICATION, POTS-ACTIONS-DIAL

IMPORT

$CALL - DEFINITIONS$
$POTS - ACTIONS - OFF - HOOK$
$POTS - ACTIONS - ON - HOOK$
$POTS - ACTIONS - COMMUNICATION$
$POTS - ACTIONS - DIAL$
$POTS - ACTIONS - CLEANING$

DECLARATIONS

bps : VARIABLE

bps is a flexible variable containing the current state of the basic system.

$ubps$: VARIABLE

ubps is a flexible variable containing the current state of the user and system

DEFINITIONS

The flexible variables are used within the invariant, permitting the strengthening of action refinements.

$FV\_INVARIANT \triangleq (ubps = (users, bps))$

The initialisation of the system.

$$POTS\_Initialisation \triangleq \begin{array}{l} \wedge\ Users\_And\_Basic\_Phone\_System(ubps) \\ \wedge\ \forall u \in Users\_Identification : ubps[u] = \emptyset \\ \wedge\ (ubps = (users, bps)) \end{array}$$

POTS_Next defines the translation relation of the global telephone system.

$POTS\_Next \triangleq$
$\exists user \in User\_Identification : \exists phone \in Phones :$
$\exists init\_user\_and\_phones\_set, dest\_user\_and\_phones\_set \in \textsc{Subset}\,(Users\_Identification \times Phones) :$
$\exists call \in Call :$
$\vee\ CONNECT(user, ubps)$
$\vee\ DISCONNECT(user, phone, ubps)$
$\vee\ OFF\_HOOK\_RINGING(user, phone, ubps)$
$\vee\ OFF\_HOOK\_CALLING(user, phone, ubps)$
$\vee\ DIAL(init\_user\_and\_phones\_set, dest\_user\_and\_phones\_set, ubps)$
$\vee\ COMMUNICATION\_OK(init\_user\_and\_phones\_set, dest\_user\_and\_phones\_set, call, ubps)$
$\vee\ COMMUNICATION\_DOWN(init\_user\_and\_phones\_set, dest\_user\_and\_phones\_set, call, ubps)$
$\vee\ COMMUNICATION\_BUSY(init\_user\_and\_phones\_set, dest\_user\_and\_phones\_set, call, ubps)$
$\vee\ ON\_HOOK(user\_and\_phone\_set, call, ubps)$
$\vee\ CLEAN\_PB(call, ubps)$
$POTS\_Fairness \triangleq$

$\forall user \in User\_Identification$:
$\forall phone \in Phones$:
$\forall init\_user\_and\_phones\_set, dest\_user\_and\_phones\_set \in \textsc{Subset}\,(Users\_Identification \times Phones)$:
$\forall call \in Call$:
$\land\ \mathrm{WF}_{ubps,bps,users}\ CONNECT(user, ubps)$
$\land\ \mathrm{WF}_{ubps,bps,users}\ DISCONNECT(user, phone, ubps)$
$\land\ \mathrm{WF}_{ubps,bps,users}\ OFF\_HOOK\_RINGING(user, phone, ubps)$
$\land\ \mathrm{WF}_{ubps,bps,users}\ OFF\_HOOK\_CALLING(user, phone, ubps)$
$\land\ \mathrm{WF}_{ubps,bps,users}\ DIAL(init\_user\_and\_phones\_set, dest\_user\_and\_phones\_set, ubps)$
$\land\ \mathrm{WF}_{ubps,bps,users}\ COMMUNICATION\_OK(init\_user\_and\_phones\_set, dest\_user\_and\_phones\_set, call, ubps)$
$\land\ \mathrm{WF}_{ubps,bps,users}\ COMMUNICATION\_DOWN(init\_user\_and\_phones\_set, dest\_user\_and\_phones\_set, call, ubps)$
$\land\ \mathrm{WF}_{ubps,bps,users}\ COMMUNICATION\_BUSY(init\_user\_and\_phones\_set, dest\_user\_and\_phones\_set, call, ubps)$
$\land\ \mathrm{WF}_{ubps,bps,users}\ ON\_HOOK(user\_and\_phone\_set, call, ubps)$
$\land\ \mathrm{SF}_{ubps,bps,users}\ CLEAN\_PB(call, ubps)$

$$POTS\_Specification \triangleq \begin{array}{l} \land\ POTS\_Initialisation \\ \land\ \Box[FV\_INVARIANT \land POTS\_Next \land FV\_INVARIANT']_{ubps} \\ \land\ POTS\_Fairness \end{array}$$

7 Feature Interaction: Call Screening and Call Forward

This feature combination is interesting because it illustrates that precision in requirements really does help to avoid feature interactions. In particular, it shows the importance of the validation process in the development of formal models. Furthermore, it illustrates why we need a mixed semantic approach.

Call Screening (CS)

Informally, call screening is easy to understand. With CS I can program my phone to ignore certain incoming calls so that my phone does not ring. The calls to be ignored are identified by a list of numbers. Every incoming call identfication is then checked against this list.

Call Forwarding (CF)

Call forwarding is also easy to understand informally. If I activate call forwarding (CF) then all my incoming calls are forwarded to another number.

The potential interaction

The question that needs to be addressed is whether there is an interaction between these two features. Informally, let us consider a simple scenario. Firstly, what happens if:

```
A screens calls from B
C forwards calls to A
B calls C
```

The problem is that A does not want to receive a call from B. However, C is visiting B and redirects calls accordingly. Now B wishes to talk to C, phones C and this results in a call to A. A therefore receives a call from B. This contradicts A's requirements for call screening. If we specify that B's call to C does not get forwarded to A then we contradict C's requirements for call forwarding. This is clearly an interaction . . . or is it? We need to analyse the problem in our formal framework.

OO LOTOS

The OO LOTOS specification lets us easily construct an executable model for the testing of our particular scenario. However, which model should we build: the one which contradicts A's requirements or the one which leaves C unsatisified? Without an explicit statement of *what* is required, we do not know which model to validate as correct.

B

The figure, below, shows the formal composition of abstract machines in B. The composition is based on the sharing of variables between machines. This makes the analysis easy to perform but the composition technique very limited. We treat each component machine as if it was the environment of the other. Then we check that all operations respect the invariants of each individual machine.

The application of such a composition technique is very useful when analysing the abstract requirements of feature compositions. We can immediately detect contradictory requirements because invariants are unprovable.

When composing call forwarding with call screening we used the B tools as follows:

- We proved the correctness of the POTS machine wrt the POTS invariant.
- We proved the correctness of the CF machine wrt the CF invariant.
- We proved the correctness of the CS machine wrt the CS invariant.
- We proved the correctness of combining the POTS machine with the CS machine, producing a new machine M3.
- We proved the correctness of combining M3 with CF.

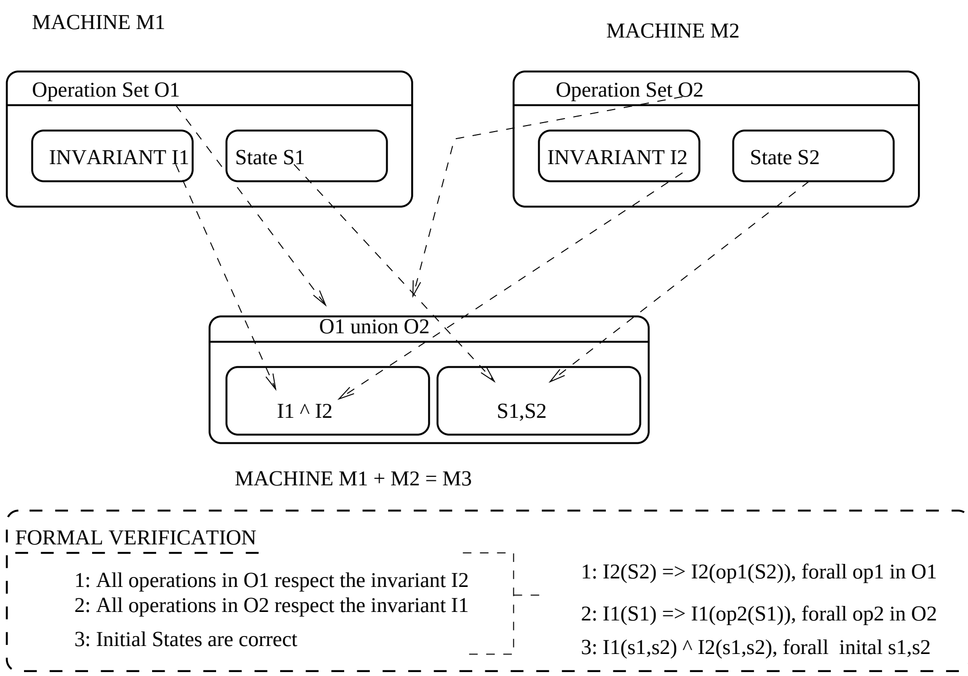

Figure 3: Machine Composition in B

At each of these stages, we had to change our specifications in response to problems with the invariant proofs. For example, we found problems in POTS when telephone users unsubscribed when in the middle of a service. In other words, using the B tools helped us to identify where our specifications were incomplete. Furthermore, in the case of the CF-CS composition we realised that there were contradictions in the requirements only when the specifications of each individual feature were not well expressed. There is in fact no interaction with the two services if we make no assumptions about who lifts a phone (A or C in our scenario are both capable of lifting the phone) when it is ringing.

The B specification led us to generate a set of requirements as a conjunction of invariant properties (which are not contradictory). We now change the LOTOS specification to remove the unnecessary assumption which we discovered using the proof tools. Then, using the OO LOTOS, we validated this behaviour through animation. (During the design process, we implemented the distinction between the two users (A and C) by having two different types of ringing at the phone, one for normal calls to A, and one for calls forwarded to C.)

TLA

The TLA is needed for one simple reason: we cannot guarantee that the services will *eventually* be carried out. The use of internal actions introduces the need for fairness requirements.

Without TLA, how can we express the fact that if a call is being forwarded it will eventually reach the new destination? We use the TLA to express such properties; although the integration of the TLA semantics with B and OO LOTOS is not complete. We are working on the notion of a *fair object* which will always guarantee that it will eventually be able to service a particular request.

We identify five different types of *client eventuality requirements*:

- **Immediately and Obliged**
  A client may require that a service request be serviced **immediately**. If it cannot be carried out then the client cannot function properly.

- **Eventually and Obliged**
  A client may require that a service is carried out **eventually**, i.e. in a finite period of time. If it is not carried out eventually then the client cannot function properly.

- **Immediately Conditional**
  A client may require a service immediately but if it cannot be done without delay then it must be informed so that it can attempt to do something else.

- **Eventually Conditional**
  The service is required eventually but if it cannot be guaranteed in a finite period of time then the client must be informed so that something else can be done.

- **Unconditional**
  The client wants the service but places no eventuality requirements on when the service must be performed.

In a *fair object* specification we must define explicitly the eventuality requirements that clients must place on servers. This covers two aspects:

- **Explicit Customer Requirements**
  This is the case when the client being considered is the complete system and the environment of the system is the customer. Eventuality requirements specified at this level correspond to *explicit fairness* properties.

- **Internal Client Requirements**
  This is the case when the internal components of the system place eventuality requirements on each other. The correct functioning of the system as seen by the customer is dependent on some hidden internal eventuality requirements being met. These requirements are generated during the system development as we move from abstract to concrete. In an unstructured system no such requirements will exist. However, as we structure the system we will introduce such requirements between the components we introduce.

It is the use of TLA semantics which make it possible to work with these eventuality concepts.

8 The Feature Case Studies: A Review

We cannot possibly review all the feature compositions which we have developed using our mixed model approach. Rather, we review the main lessons that we learned from our case studies:

Use Strong Typing

The advantages of typing in all forms of development are well known. Types are not needed in correct systems but they do help to create correct systems. A simple telephone example is that of the concept of a telephone number. Clearly, telephone numbers are polymorphic within a concrete subtyping hierarchy. Without types, it is difficult to handle the different natures of telephone numbers at the requirements stage.

Use Types as Behaviours or Roles

Here we impose the object oriented principle of classification. The class is used as the fundamental unit of behaviour (and as the means of typing these units). Thus, every feature is a class which plays a specific role. The class hierarchy provides a behavioural benchmark for categorising features. Already some work has been done towards creating a benchmark for feature interactions [7] and we believe an object oriented strategy would complement this research.

Use Invariants

Invariants are used to define relationships between components of a system that must be true during the lifetime of the system. They are a well understood, formal means of specifying requirements in a compositional manner. Every non trivial component (of a (sub)system), i.e. one with its own components, has an associated invariant and there is one invariant between all components (of a (sub)system).

Avoid ambiguity in naming of actions

The practice of polymorphic actions arises from the minimilistic interface provided by most telephones. For example, a *flash hook* action can signal different things to different features. If these features are requested at the same time then the meaning of a *flash hook* may be ambiguous. Whenever possible try to maintain a clear distinction between *abstract actions/signals* in the requirements model and *concrete actions/signals* in the implementation model.

Features should be explicit not implicit

Typically, features appear in formal specifications only as implicit, derivable properties of the total system. We can verify that a system complies to the requirements of a feature but there is no compositional means of removing the feature and examining it as a single identity. Given a logical landscape as our semantic basis would permit such a compositional view since the properties required of a feature are exactly its specification. However, in such a logical approach,

the actual development of features is much more complicated. We believe that we should have an *explicit* compositional approach in the same spirit as the logical method, whilst avoiding the synthesis and analysis problems that arise from features that are specified logically.

Arbitration and interaction resolution

We have not examined the question of interaction resolution in this paper, since we wanted to concentrate on the question of interaction avoidance arising from formal feature requirements models. However, interactions do occur when feature requirements are contradictory. Clearly, when two features have contradictory requirements then one (or both) of the feature specifications will have to change if the two features are to be combined in a meaningful way. Arbitration is one means of changing the requirements of some feature(s) to remove the contradiction. This arbitration can be done both statically, in the specification, or dynamically, during execution. Arbitration can be done independently of the order in which services are implemented. The process of arbitration is one which we are currently examining.

Choosing Modelling Languages

The advantage of a logical approach is self evident: the combination of features is just logical conjunction and the absence of interaction is automatic, provided the result is not false. The disadvantage of such an approach is the difficulty in constructing new features and analysing their dynamic behaviour. The principal problem is that of communicating with the clients in such a mathematical model. Contrastingly, more operational (state based) approaches are more powerful with respect to synthesis and analysis. However, it is then much more difficult to say what it means for two features to be contradictory (i.e. have requirements that cannot be met at the same time). The main source of feature interaction problems seems to be exceptions. Using invariants helps to transfer the analysis of exceptions away from the dynamic and towards a purely static approach. An object oriented approach gives us abstraction and generalisation within a compositional user friendly framework. A combination of a number of semantic frameworks seems to be the only option for all our modelling needs. The problem of feature interaction is so general, with complex, diverse issues, that we cannot expect a single semantic model approach to be satisfactory.

Operational Requirements are necessary

Concurrent (independent) development of features requires separation of specification from implementation. However, for implementors (designers) to fully understand requirements we expect to be able to animate our specifications. This is also a necessary part of validation. Thus we require operational semantics [30] for our feature specifications (as well as our more abstract logical requirements for testing compatibility).

The incremental development problem: minimise impact of change

In traditional problem domains (and using state-of-the-art development methods) when new functionality is added to a system it is possible to do this by *connecting it* to only a small subset of the system components. (We will not for now attempt to define the different types of connec-

tion.) Additions that are localised (with fewer connections) are easier to make than those which are spread about the system. The addition of a feature is inherently a non-local problem (in the current telephone architectures) because it necessitates connection with most of the other components (the other features) in the system. Hence, each addition has global impact. We are searching for an architecture which supports local incrementation techniques.

Restrictive Assumption Approach

Restrict the assumptions that a feature developer can make about the behaviour of its environment (other features in the system included). Since new features will be added later we cannot place any assumptions on them. However, in some cases assumptions must be made. These should be specified as invariant properties that are amenable to static analysis. Our goal is to simplify this analysis by formalising a minimum assumption set that does not restrict our functionality but does eliminate interactions.

9 Conclusions

We have shown the need for different semantics for:

- **Structuring a model** — where we apply object oriented conceptualisation.
- **Validating a model** — where we use the OO LOTOS
- **Verifying invariant properties** — where we use B
- **Verifying liveness and fairness properties** — where we use TLA.
- **Performing verifiable refinements** — where we use TLA

Currently, we are attempting to integrate the different semantics into one coherent model [15, 14, 18].

A mixed-semantics approach is particularly important in telephone feature development. However, we believe our approach is also applicable in all other problem domains where functionality is distributed over concurrent resources.

References

[1] J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.

[2] B-core. B-Toolkit User's Manual, Release 3.2. Technical report, B-core, 1996.

[3] T. Bolognesi and M. Caneve. SQUIGGLES: A tool for the analysis of LOTOS specifications. In K.J.T. Turner, editor, *The 2nd International Conference on Formal Description Techniques (FORTE 89)*, 1989.

[4] G. Booch. *Object oriented design with applications*. Benjamin Cummings, 1991.

[5] L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions In Telecommunications*. IOS Press, 1994.

[6] E. Brinksma and Scollo G. Formal notions of implementation and conformance in lotos. Mem: INT-86-13, University of Twente, NL, December 1986.

[7] E. J. Cameron, N. D. Griffeth, Y. Lin, M. E. Nilson, and W. K. Schnure. A feature interaction benchmark for in and beyond. In *Feature Interactions In Telecommunications*, 1994.

[8] K. E. Cheng and T. Ohta, editors. *Feature Interactions In Telecommunications III*. IOS Press, 1995.

[9] P. Coad and E. Yourdon. *Object oriented analysis*. Prentice-Hall (Yourdon Press), 1990.

[10] L. Constantine. Beyond the madness of methods: System structure methods and converging design. In *Software Development 1989*. Miller-Freeman, 1989.

[11] Geoff Cutts. *Structured system analysis and design method*. Blackwell Scientific Publishers, 1991.

[12] Digilog. Atelier B, Guide de l'utilisateur v2.0. Technical report, Digilog, 1995.

[13] H. Ehrig and Mahr B. *Fundamentals of Algebraic Specification I*. Springer-Verlag, Berlin, 1985. EATCS Monographs on Theoretical Computer Science (6).

[14] J.-P. Gibson and D. Méry. A Unifying Model for Multi-Semantic Software Development. Rapport Interne CRIN-96-R-110, Linz (Austria), July 1996.

[15] J.-P. Gibson and D. Méry. A Unifying Model for Specification and Design. Rapport Interne CRIN-96-R-110, Linz (Austria), July 1996.

[16] J. Paul Gibson. Formal object based design in LOTOS. Tr-113, University of Stirling, Computing Science Department, Stirling, Scotland, 1994.

[17] J.Paul Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS*. Tech. report csm-114, Stirling University, August 1993.

[18] J.Paul Gibson and D. Mery. Telephone feature verification: Translating SDL to TLA+. Accepted for SDL97, Evry, France (to be announced), 1997.

[19] R. Guillemot, M. Haj-Hussein, and L. Logroppo. Executing large LOTOS specifications. In *Proceedings of Prototyping, Specification, Testing and Verification VIII*. North-Holland, 1991.

[20] C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[21] ISO. LOTOS — a formal description technique based on the temporal ordering of observed behaviour. Technical report, International Organisation for Standardisation IS 8807, 1988.

[22] C. B. Jones and R. C. Shaw. *Case Studies in Systematic Software Development*. Prentice-Hall International Series in Computer Science. Prentice-Hall, 1990. ISBN0-13-116088-5.

[23] L. Lamport. A temporal logic of actions. Technical Report 57, DEC Palo Alto, april 1990.

[24] L. Lamport. A temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[25] L. Lamport. TLA$^+$. Technical report, December, 5th july 1995.

[26] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[27] K. Ohmaki, K. Futatsugi, and K. Takahashi. A basic LOTOS simulator in OBJ. Computer Language Section, Computer Science Division, Electrotechnical Laboratory, 1-1-4 Umezono, Japan, Draft Report, 1990.

[28] J. M. Spivey. *Understanding Z : a specification language and its formal semantics*. Cambridge University Press, 1987.

[29] van Eijk, Vissers, and Diaz. *The Formal Description Technique LOTOS*. North-Holland, Amsterdam, 1989.

[30] Pamela Zave. The operational versus the conventional approach to software development. *Comm. ACM*, 27:104–118, 1984.

[31] Pamela Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer Magazine*, pages 18–23, August 1993.